Program and Programming Style

The material for this lecture is drawn, in part, from
The Practice of Programming (Kernighan & Pike) Chapter 1

For Your Amusement

“Any fool can write code that a computer can understand. Good programmers write code that humans can understand.” -- Martin Fowler

“Good code is its own best documentation. As you’re about to add a comment, ask yourself, ‘How can I improve the code so that this comment isn’t needed?’” -- Steve McConnell

“Programs must be written for people to read, and only incidentally for machines to execute.” -- Abelson / Sussman

“Everything should be built top-down, except the first time.” - Alan Perlis

“Programming in the Large” Steps

Design & Implement
- Program & programming style <-- we are here
- Common data structures and algorithms
- Modularity
- Building techniques & tools (done)

Debug
- Debugging techniques & tools

Test
- Testing techniques (done)

Maintain
- Performance improvement techniques & tools

Goals of this Lecture

Help you learn about:
- Good program style
- Good programming style

Why?
- A well-styled program is more likely to be correct than a poorly-styled program
- A well-styled program is more likely to stay correct (i.e. is more maintainable) than a poorly-styled program
- A power programmer knows the qualities of a well-styled program, and how to compose one quickly

Agenda

Program style
- Qualities of a good program

Programming style
- How to write a good program quickly

Motivation for Program Style

Who reads your code?
- The compiler
- Other programmers

This is a working ray tracer! (courtesy of Paul Heckbert)
Motivation for Program Style

Why does program style matter?
- Correctness
  - The clearer a program is, the more likely it is to be correct
- Maintainability
  - The clearer a program is, the more likely it is to stay correct over time

Good program ≈ clear program

Choosing Names

Use descriptive names for globals and functions
- E.g. `display`, `CONTROL`, `CAPACITY`

Use concise names for local variables
- E.g., `i` (not `arrayIndex`) for loop variable

Use case judiciously
- E.g., `Stack_push` (Module function)
  - `CAPACITY` (constant)
  - `buf` (local variable)

Use a consistent style for compound names
- E.g., `frontsize`, `frontSize`, `front_size`

Use active names for functions that do something
- E.g., `getchar()`, `putchar()`, `Check_octal()`, etc.

Not necessarily for functions that are something: `sin()`, `sqrt()`

Using C Idioms

Use C idioms
- Example: Set each array element to 1.0.
- Bad code (complex for no obvious gain)

```c
i = 0;
while (i <= n-1)
array[i++] = 1.0;
```

- Good code (not because it’s vastly simpler—it isn’t!—but because it uses a standard idiom that programmers can grasp at a glance)

```c
for (i=0; i<n; i++)
array[i] = 1.0;
```

- Don’t feel obliged to use C idioms that decrease clarity

Revealing Structure: Expressions

Use natural form of expressions
- Example: Check if integer `n` satisfies `j < n < k`
- Bad code

```c
if (!(n >= k) && !(n <= j))
```

- Good code

```c
if ((j < n) && (n < k))
```

Conditions should read as you’d say them aloud
- Not “Conditions shouldn’t read as you’d never say them in other than a purely internal dialog!”

Parenthesize to resolve ambiguity
- Example: Check if integer `n` satisfies `j < n < k`

- Common code

```c
if (j < n && n < k)
```

- Clearer code (maybe)

```c
if ((j < n) && (n < k))
```

It’s clearer depending on whether your audience can be trusted to know the precedence of all the C operators. Use your judgment on this!

Parenthesize to resolve ambiguity (cont.)
- Example: Read and print character until end-of-file

- Bad code

```c
while ((c = getchar()) != EOF)
putchar(c);
```

- Good-ish code

```c
while ((c = getchar()) != EOF)
putchar(c);
```

- (Code with side effects inside expressions is never truly “good”, but at least this code is a standard idiomatic way to write it in C)
Revealing Structure: Expressions

Break up complex expressions
- Example: Identify chars corresponding to months of year
- Bad code

```java
if (c == 'J' || c == 'Y' || c == 'M' ||
    c == 'A' || c == 'S' || c == 'O' ||
    c == 'N' || c == 'D'))
    do_this();
else do_that();
```

- Good code – lining up things helps

```java
if (c == 'J' || c == 'Y' || c == 'M') {
    switch (c) {
    case 'J': case 'Y': case 'M':
        case 'A': case 'S': case 'O':
        case 'N': case 'D':
            do_this();
            break;
            default:
            do_that();
    }
}
```

- Very common, though, to elide parentheses

```java
if (c == 'J' || c == 'Y' || c == 'M') {
    switch (c) {
    case 'J': case 'Y': case 'M':
        case 'A': case 'S': case 'O':
        case 'N': case 'D':
            do_this();
            break;
            default:
            do_that();
    }
}
```

Revealing Structure: Spacing

Use readable/consistent spacing
- Example: Assign each array element a[j] to the value j.
- Bad code

```java
for (j=0; j<100; j++) a[j]=j;
```

- Good code

```java
for (j = 0; j < 100; j++)
a[j] = j;
```

- Often can rely on auto-indenting feature in editor

Revealing Structure: Indentation

Use readable/consistent/correct indentation
- Example: Checking for leap year (does Feb 29 exist?)

```java
legal = TRUE;
if (month == FEB) {
    if ((year % 4) == 0) {
        if (day > 29)
            legal = FALSE;
    } else
    if (day > 28)
        legal = FALSE;
}
else
    legal = FALSE;
```

Revealing Structure: “Paragraphs”

Use “else-if” for multi-way decision structures
- Example: Comparison step in a binary search.
- Bad code

```java
if (x < a[mid])
    high = mid - 1;
else
    if (x > a[mid])
        low = mid + 1;
    else
        return mid;
```

- Good code

```java
if (x < a[mid])
    high = mid - 1;
else
    if (x > a[mid])
        low = mid + 1;
    else
        return mid;
```

Use blank lines to divide the code into key parts

```c
#include <stdio.h>
#include <stdlib.h>

/* Read a circle’s radius from stdin, and compute and write its diameter and circumference to stdout. Return 0 if successful. */
int main(void)
{
    const double PI = 3.14159;
    int radius;
    int diam;
    double circum;
    printf("Enter the circle’s radius:\n");
    if (scanf("%d", &radius) != 1) {
        fprintf(stderr, "Error: Not a number\n");
        exit(EXIT_FAILURE);
    }
    diam = 2 * radius;
    circum = 2 * PI * radius;
    printf("\nDiameter = \%d\nCircumference = \%f\n", diam, circum);
    return 0;
}
```
Revealing Structure: “Paragraphs”

Use blank lines to divide the code into key parts

```c
# include <stdio.h>
# include <stdlib.h>

/* Compute the diameter and circumference. */
# define PI 3.14159;
int main(void)
{ int radius;
  double diam;
  double circun;
  /* Read the circle’s radius. */
  printf("Enter the circle’s radius:\n");
  if (scanf("%d", &radius) != 1) {
    printf(stderr, "Error: Not a number!\n");
    exit(EXIT_FAILURE); /* or: return EXIT_FAILURE; */
    return 0;
  }
  diam = 2 * radius;
  circun = PI * (double)diam;
  printf("A circle with radius %d has diameter %d\n", radius, diam);
  printf("and circumference %f.\n", circun);
  return 0;
}
```

Composing Comments

- Master the language and its idioms
  - Let the code speak for itself
  - And then...
- Compose comments that add new information
  ```c
  i++;  // Add one to i. */
  ```
- Comment paragraphs of code, not lines of code
  - E.g., “Sort array in ascending order”
- Comment global data
  - Global variables, structure type definitions, field definitions, etc.
- Compose comments that agree with the code!!!
  - And change as the code itself changes!!!

Composing Comments

Comment sections (“paragraphs”) of code, not lines of code

```c
# include <stdio.h>
# include <stdlib.h>

/* Read a circle’s radius from stdin, and compute and write its
diameter and circumference to stdout. Return 0 if successful. */
int main(void)
{ const double PI = 3.14159;
  int radius;
  int diam;
  double circun;
  /* Read the circle’s radius. */
  printf("Enter the circle’s radius:\n");
  if (scanf("%d", &radius) != 1) {
    printf(stderr, "Error: Not a number!\n");
    exit(EXIT_FAILURE); /* or: return EXIT_FAILURE; */
    return 0;
  }
  diam = 2 * radius;
  circun = PI * (double)diam;
  /* Print the results. */
  printf("A circle with radius %d has diameter %d\n", radius, diam);
  printf("and circumference %f.\n", circun);
  return 0;
}
```

Composing Function Comments

- **Describe what a caller needs to know to call the function properly**
  - Describe what the function does, not how it works
  - Code itself should clearly reveal how it works...
  - If not, compose “paragraph” comments within definition
- **Describe input**
  - Parameters, files read, global variables used
- **Describe output**
  - Return value, parameters, files written, global variables affected
- **Refer to parameters by name**

Composing Function Comments

**Bad function comment**

```c
/* decomment.c */
/* Read a character. Based upon the character and the current DFA state, call the appropriate
state-handling function. Repeat until end-of-file. */
int main(void)
{ ... }
```

Describes how the function works
Composing Function Comments

Good function comment

```c
/* decomment.c */
/* Read a C program from stdin. Write it to stdout with each comment replaced by a single space. Preserve line numbers. Return 0 if successful, EXIT_FAILURE if not. */
int main(void)
{
    ...
}
```

• Describes what the function does

Using Modularity

Abstraction is the key to managing complexity
• Abstraction is a tool (the only one???) that people use to understand complex systems
• Abstraction allows people to know what a (sub)system does without knowing how

Proper modularity is the manifestation of abstraction
• Proper modularity makes a program’s abstractions explicit
• Proper modularity can dramatically increase clarity
• ⇒ Programs should be modular

However
• Excessive modularity can decrease clarity!
• Improper modularity can dramatically decrease clarity!!!
• ⇒ Programming is an art

Modularity Examples

Examples of function-level modularity
• Character I/O functions such as getchar() and putchar()
• Mathematical functions such as sin() and cos()
• Function to sort an array of integers

Examples of file-level modularity
• (See subsequent lectures)

Program Style Summary

Good program = clear program

Qualities of a clear program
• Uses appropriate names
• Uses common idioms
• Reveals program structure
• Contains proper comments
• Is modular

Agenda

Program style
• Qualities of a good program

Programming style
• How to write a good program quickly

Bottom-Up Design

Bottom-up design 😊
• Design one part of the system in detail
• Design another part of the system in detail
• Combine
• Repeat until finished

Bottom-up design in painting
• Paint part of painting in complete detail
• Paint another part of painting in complete detail
• Combine
• Repeat until finished
• Unlikely to produce a good painting
(except sometimes: see the movie “Tim’s Vermeer”)
Bottom-Up Design

Bottom-up design in programming
- Compose part of program in complete detail
- Compose another part of program in complete detail
- Combine
- Repeat until finished
- Unlikely to produce a good program

Top-Down Design

Top-down design
- Design entire product with minimal detail
- Successively refine until finished

Top-down design in painting
- Sketch the entire painting with minimal detail
- Successively refine until finished

Top-Down Design

Top-down design in programming
- Define main() function in pseudocode with minimal detail
- Refine each pseudocode statement
  - Small job ⇒ replace with real code
  - Large job ⇒ replace with function call
- Repeat in (mostly) breadth-first order until finished
- Bonus: Product is naturally modular

Example: Text Formatting

Functionality (derived from King Section 15.3)
- Input: ASCII text, with arbitrary spaces and newlines
- Output: the same text, left and right justified
- Fit as many words as possible on each 50-character line
- Add even spacing between words to right justify the text
- No need to right justify last line
- Assumptions
  - "Word" is a sequence of non-white-space chars followed by a white-space char or end-of-file
  - No word is longer than 20 chars

Example Input and Output

"C is quirky, flawed, and an enormous success. While accidents of history surely helped, it evidently satisfied a need for a system implementation language efficient enough to displace assembly language, yet sufficiently abstract and fluent to describe algorithms and interactions in a wide variety of environments." — Dennis Ritchie

"C is quirky, flawed, and an enormous success. While accidents of history surely helped, it evidently satisfied a need for a system implementation language efficient enough to displace assembly language, yet sufficiently abstract and fluent to describe algorithms and interactions in a wide variety of environments." — Dennis Ritchie
Caveats

Caveats concerning the following presentation
- Function comments and some blank lines are omitted
- Because of space constraints
- Don't do that!!!
- Design sequence is idealized
- In reality, typically much backtracking would occur

```c
enum {MAX_WORD_LEN = 20};
int main(void)
{
    char word[MAX_WORD_LEN+1];
    int wordLen;
    <clear line>
    wordLen = readWord(word);
    while (<there is a word>)
    {
        if (<word doesn't fit on line>)
        {
            <write justified line>
            <clear line>
        }
        <add word to line>
        wordLen = readWord(word);
    }
    if (<line isn't empty>)
    <write line>
    return 0;
}
```
The main() Function

```c
enum {MAX_WORD_LEN = 20};
enum {MAX_LINE_LEN = 50};
int main(void)
{
    char word[MAX_WORD_LEN+1];
    char line[MAX_LINE_LEN+1];
    int wordLen;
    int lineLen;
    <clear line>
    wordLen = readWord(word);
    while (wordLen != 0)
    {
        if (word doesn't fit on line)
            <write justified line>
            <clear line>
        lineLen = addWord(word, line, lineLen);
        wordLen = readWord(word);
    }
    if (lineLen > 0)
        puts(line);
    return 0;
}
```

The readWord() Function

```c
int readWord(char *word)
{
    <skip over white space>
    <read chars, storing up to MAX_WORD_LEN in word>
    <return length of word>
}
```
The readWord() Function

```c
int readWord(char *word)
{
    int ch;
    /* Skip over white space. */
    ch = getchar();
    while ((ch != EOF) && isspace(ch))
        ch = getchar();
    /* read up to MAX_WORD_LEN chars into word */
    <read up to MAX_WORD_LEN chars into word>
    "<return length of word>"
}
```

Note the use of a function from the standard library. Very appropriate for your top-down design to target things that are already built.

The readWord() Function

```c
int readWord(char *word)
{
    int ch;
    int pos = 0;
    ch = getchar();
    /* Skip over white space. */
    while ((ch != EOF) && isspace(ch))
        ch = getchar();
    /* Read up to MAX_WORD_LEN chars into word. */
    while ((ch != EOF) && (!isspace(ch)))
    { if (pos < MAX_WORD_LEN)
        { word[pos] = (char)ch;
          pos++;
        }
    ch = getchar();
    word[pos] = ' ';
    return pos;
}
```

readWord() gets away with a character past the end of the word.

The addWord() Function

```c
int addWord(const char *word, char *line, int lineLen)
{
    /* if line already contains words, then append a space. */
    if (newLineLen > 0)
    { strcat(line, " ");
        newLineLen++;
    }
    <append word to line>
    <return the new line length>
}
```

The addWord() Function

```c
int addWord(const char *word, char *line, int lineLen)
{
    int newLineLen = lineLen;
    /* if line already contains words, then append a space. */
    if (newLineLen > 0)
    { strcatLine(" ");
        newLineLen++;
    }
    <append word to line>
    <return the new line length>
}
The addWord() Function

```c
int addWord(const char *word, char *line, int lineLen)
{
    int newLineLen = lineLen;
    /* If line already contains words, then append a space. */
    if (newLineLen > 0)
    { strcat(line, " ");
        newLineLen++;
    } strcat(line, word);
    return newLineLen;
}
```

The addWord() Function

```c
int addWord(const char *word, char *line, int lineLen)
{
    int newLineLen = lineLen;
    /* If line already contains some words, then append a space. */
    if (newLineLen > 0)
    { strcat(line, " ");
        newLineLen++;;
    } strcat(line, word);
    return newLineLen;
}
```

Status

```
main

readWord  writeLine  addWord
```

The writeLine() Function

```c
void writeLine(const char *line, int len, int wordCount)
{
    int i, extraSpaces;
    /* Compute number of extra spaces for line. */
    extraSpaces = MAX_LINE_LEN - len;
    for (i = 0; i < len; i++)
    { if (line[i] == ' ')
            putchar(line[i]);
        else
        { <compute additional spaces to insert>
            <print a space, plus additional spaces>
            <decrease extra spaces and word count>
        }
    } putchar(\n');
}
```

The writeLine() Function

```c
void writeLine(const char *line, int len, int wordCount)
{
    int i, extraSpaces, spacesToInsert;
    /* Compute number of extra spaces for line. */
    extraSpaces = MAX_LINE_LEN - len;
    for (i = 0; i < len; i++)
    { if (line[i] == ' ')
            putchar(line[i]);
        else
        { /* Compute additional spaces to insert. */
            spacesToInsert = extraSpaces / (wordCount - 1);
            <print a space, plus additional spaces>
            <decrease extra spaces and word count>
        }
            putchar(\n');
}
```
The writeLine() Function

```c
void writeLine(const char *line, int lineLen, int wordCount)
{
    int i, extraspace, spacesToInsert, j;

    /* Compute number of extra spaces for line. */
    extraspace = MAX_LINE_LEN - lineLen;
    for (i = 0; i < lineLen; i++)
    { if (line[i] == ' ')
        putchar(line[i]);
        else
        { /* Compute additional spaces to insert. */
          spacesToInsert = extraspace / (wordCount - 1);

          /* Print a space, plus additional spaces. */
          for (j = 1; j <= spacesToInsert + 1; j++)
          putchar(' ');
        }
    }
    putchar('
');
}
```

Example: If extraspace is 10 and wordCount is 5, then gaps will contain 2, 2, 3, and 3 extra spaces respectively.

Status

```
main
```

```
readWord
```

```
writeLine
```

```
addWord
```

Complete!

Top-Down Design and Modularity

Note: Top-down design naturally yields modular code

Much more on modularity in upcoming lectures

Aside: Least-Risk Design

Design process should minimize risk

Bottom-up design
- Compose each child module before its parent
- Risk level: high
  - May compose modules that are never used

Top-down design
- Compose each parent module before its children
- Risk level: low
  - Compose only those modules that are required

Least-risk design
- The module to be composed next is the one that has the most risk
- The module to be composed next is the one that, if problematic, will require redesign of the greatest number of modules
- The module to be composed next is the one that poses the least risk of needing to redesign other modules
- The module to be composed next is the one that poses the least risk to the system as a whole
- Risk level: minimal (by definition)
Aside: Least-Risk Design

Recommendation
- Work mostly top-down
- But give high priority to risky modules
- Create scaffolds and stubs as required

Summary

Program style
- Choose appropriate names (for variables, functions, …)
- Use common idioms (but not at the expense of clarity)
- Reveal program structure (spacing, indentation, parentheses, …)
- Compose proper comments (especially for functions)
- Use modularity (because modularity reveals abstractions)

Programming style
- Use top-down design and successive refinement
- But know that backtracking inevitably will occur
- And give high priority to risky modules

Are we there yet?

Now that the top-down design is done, and the program “works,” does that mean we’re done?

No. There are almost always things to improve, perhaps by a bottom-up pass that better uses existing libraries.

The second time you write the same program, it turns out better.

What’s wrong with this output?

"C is quirky, flawed, and an enormous success. While accidents of history surely helped, it evidently satisfied a need for a system implementation language efficient enough to displace assembly language, yet sufficiently abstract and fluent to describe algorithms and interactions in a wide variety of environments."  -- Dennis Ritchie

"C is quirky, flawed, and an enormous success. While accidents of history surely helped, it evidently satisfied a need for a system implementation language efficient enough to displace assembly language, yet sufficiently abstract and fluent to describe algorithms and interactions in a wide variety of environments."  -- Dennis Ritchie

What’s better with this output?

"C is quirky, flawed, and an enormous success. While accidents of history surely helped, it evidently satisfied a need for a system implementation language efficient enough to displace assembly language, yet sufficiently abstract and fluent to describe algorithms and interactions in a wide variety of environments.”  -- Dennis Ritchie

Challenge problem

Design a function  

```c
int spacesHere(int i, int k, int n)
```

that calculates how many marbles to put into the i'th jar, assuming that there are n marbles to distribute over 4 jars.

(1) The jars should add up to n, that is,  

```c
s=0; for(i=0;i<k;i++) s+=spacesHere(i,k,n); assert (s==n);
```

or in math notation,  

```
\sum_{i=0}^{k} \text{spacesHere}(i,k,n) = n
```

(2) Marbles should be distributed evenly—the “extra” marbles should not bunch up in nearby jars.

HINT: You should be able to write this in one or two lines, without any loops.

My solution used floating-point division and rounding; do “man round” and pay attention to where that man page says “include <math.h>”.
Appendix: The “justify” Program

```c
#include <stdio.h>
#include <ctype.h>
#include <string.h>

enum {MAX_WORD_LEN = 50};
enum {MAX_LINE_LEN = 50};

/* Append word to line, making sure that the words within line are
separated with spaces. Return new line length. */
int addWord(char *word, char *line, int linelen)
{
    int newlinelen = linelen;
    /* If line already contains some words, then append a space. */
    if (newlinelen > 0)
    {
        strcat(line, " ");
        newlinelen++;
    } else
    {
        newlinelen = strlen(word);
    }
    strcat(line, word);
    newlinelen += strlen(word);
    return newlinelen;
}

/* Read words from stdin, and write the words in justified format to stdout. Return 0. */
int main(void)
{
    /* Simplifying assumptions:
Each word ends with a space, tab, newline, or end-of-file.
No word is longer than MAX_WORD_LEN characters. */
    char word[MAX_WORD_LEN + 1];
    char line[MAX_LINE_LEN + 1];
    int wordlen;
    int linelen = 0;
    int wordcount = 0;
    line[0] = '\0';
    linelen = 0;
    wordcount = 0;
    /* Read a word from stdin. Assign it to word. Return the length of
the word, or 0 if no word could be read. */
    int readWord(char *word)
    { int ch, pos = 0;
        /* Skip over white space. */
        ch = getchar();
        while ((ch = getchar()) == (int) ' ')
            ch = getchar();
        /* Store chars up to MAX_WORD_LEN in word. */
        while (ch && (ch != EOF) && (ch != (int) ' '))
        { if (linelen < MAX_WORD_LEN)
            { word[pos] = (char) ch;
                pos++;
            }
            word[pos] = ' ';
        } /* Return length of word. */
        return pos;
    }
    /* Write line to stdout, in right justified form. linelen
indicates the number of characters in line. wordcount indicates
the number of words in line. */
    void writeLine(char *line, int linelen, int wordcount)
    { int extraSpaces, spaceToInsert, i, j;
        /* Compute number of extra spaces for line. */
        extraSpaces = MAX_LINE_LEN - linelen;
        for (i = 0, i < linelen, i++)
        { if (line[i] == ' ')
            { printf("\n");
                extraSpaces -= wordcount;
            }
            else
            { /* Compute additional spaces to insert */
                spaceToInsert = extraSpaces / (wordcount - 1);
                /* Print a space, plus additional spaces. */
                for (j = 0, j < spaceToInsert + 1, j++)
                    putchar(' '); // Print a space
                /* Decrease extra spaces and word count. */
                extraSpaces = extraSpaces - 1;
                wordcount--;
            }
            printf("\n");
        }
    }
}
```

Appendix: The “justify” Program

```
```
Debugging (Part 1)

“When debugging, novices insert corrective code; experts remove defective code.”
-- Richard Pattis

“If debugging is the act of removing errors from code, what’s programming?”
-- Tom Gilb

“Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.”
-- Brian Kernighan

For Your Amusement

The first computer bug (found in the Harvard Mark II computer)

“Programming in the Large” Steps

Design & Implement
- Program & programming style (done)
- Common data structures and algorithms
- Modularity
- Building techniques & tools (done)

Test
- Testing techniques (done)

Debug
- Debugging techniques & tools <-- we are here

Maintain
- Performance improvement techniques & tools

Goals of this Lecture

Help you learn about:
- Strategies and tools for debugging your code

Why?
- Debugging large programs can be difficult
- A power programmer knows a wide variety of debugging strategies
- A power programmer knows about tools that facilitate debugging
- Debuggers
- Version control systems

Testing vs. Debugging

Testing
- What should I do to try to break my program?

Debugging
- What should I do to try to fix my program?
**Agenda**

(1) Understand error messages  
(2) Think before writing  
(3) Look for familiar bugs  
(4) Divide and conquer  
(5) Add more internal tests  
(6) Display output  
(7) Use a debugger  
(8) Focus on recent changes

---

**Understand Error Messages**

Debugging at build-time is easier than debugging at run-time, if and only if you...

Understand the error messages!

```c
#include <stdio.h>
/* Print "hello, world" to stdout and return 0. */
int main(void)
{   printf("hello, world\n");
    return 0;
}
```

What are the errors? (No fair looking at the next slide!)

---

```
#include <stdio.h>
/* Print "hello, world" to stdout and return 0. */
int main(void)
{   printf("hello, world\n");
    return 0;
}
```

What are the errors? (No fair looking at the next slide!)

---

```
#include <stdio.h>
/* Print "hello, world" to stdout and return 0. */
int main(void)
{   printf("hello, world\n");
    return 0;
}
```

What are the errors? (No fair looking at the next slide!)

---

```
#include <stdio.h>
/* Print "hello, world" to stdout and return 0. */
int main(void)
{   printf("hello, world\n");
    return 0;
}
```

What are the errors? (No fair looking at the next slide!)
Understand Error Messages

```c
#include <stdio.h>

int main(void)
{
    printf("hello, world\n")
    return 0;
}
```

$ gcc hello.c -o hello

hello.c: In function 'main':
hello.c:5: warning: implicit declaration of function 'printf'
/tmp/ccL8PMTR.o: In function 'main':
hello.c:(.text+0x1a): undefined reference to 'printf'
collect2: ld returned 1 exit status

```
#include <stdio.h>
enum StateType
{
    STATE_REGULAR,
    STATE_INWORD
};

int main(void)
{
    printf("just hanging around\n");
    return EXIT_SUCCESS;
}
```

What are the errors? (No fair looking at the next slide!)

Understand Error Messages

```c
#include <stdio.h>

// Print "hello, world" to stdout and
return 0;

int main(void)
{
    printf("hello, world\n")
    return 0;
}
```

What does this error message even mean?

```c
#include <stdio.h>
#include <stdlib.h>
enum StateType
{
    STATE_REGULAR,
    STATE_INWORD
};

int main(void)
{
    printf("just hanging around\n");
    return EXIT_SUCCESS;
}
```

$ gcc hello.c -o hello

hello.c: In function 'main':
hello.c:5: warning: two or more data types in declaration specifiers
hello.c:7: warning: return type of 'main' is not 'int'

Understand Error Messages

Caveats concerning error messages
- Line # in error message may be approximate
- Error message may seem nonsensical
- Compiler may not report the real error

Tips for eliminating error messages
- Clarity facilitates debugging
- Make sure code is indented properly
- Look for missing semicolons
- At ends of structure type definitions
- At ends of function declarations
- Work incrementally
- Start at first error message
- Fix, rebuild, repeat

Agenda

1. Understand error messages
2. Think before writing
3. Look for familiar bugs
4. Divide and conquer
5. Add more internal tests
6. Display output
7. Use a debugger
8. Focus on recent changes

Think Before Writing

Inappropriate changes could make matters worse, so...

Think before changing your code
- Explain the code to:
  - Yourself
  - Someone else
  - A Teddy bear?
- Do experiments
  - But make sure they’re disciplined
Agenda

(1) Understand error messages
(2) Think before writing
(3) Look for common bugs
(4) Divide and conquer
(5) Add more internal tests
(6) Display output
(7) Use a debugger
(8) Focus on recent changes

Look for Common Bugs

Some of our favorites:

```c
int i;
scanf("%d", i);
char c;

while (c = getchar() != EOF)
    printf("%d\n", i);
```

```c
for (i = 0; i < 10; i++)
    for (j = 0; j < 10; j++)
        printf("%d", i);
```

```
if (i = 5)
```

```
if (5 < i < 10)
```

```
if (i & 3)
```

```
What are the errors?
```

Look for Common Bugs

Some of our favorites:

```c
int i;
...
i = 3;
if (something)
    { int i;
        ...
i = 6;
    }
    printf("%d\n", i);
```

```
What value is written if this statement is present? Absent?
```

Agenda

(1) Understand error messages
(2) Think before writing
(3) Look for common bugs
(4) Divide and conquer
(5) Add more internal tests
(6) Display output
(7) Use a debugger
(8) Focus on recent changes

Divide and Conquer

Divide and conquer: To debug a program...

- Incrementally find smallest input file that illustrates the bug

- Approach 1: Remove input
  - Start with file
  - Incrementally remove lines until bug disappears
  - Examine most-recently-removed lines

- Approach 2: Add input
  - Start with small subset of file
  - Incrementally add lines until bug appears
  - Examine most-recently-added lines
**Divide and Conquer**

Divide and conquer: To debug a module...

- Incrementally find smallest client code subset that illustrates the bug
- Approach 1: Remove code
  - Start with test client
  - Incrementally remove lines of code until bug disappears
  - Examine most-recently-removed lines
- Approach 2: Add code
  - Start with minimal client
  - Incrementally add lines of test client until bug appears
  - Examine most-recently-added lines

**Add More Internal Tests**

(5) Add more internal tests

- Internal tests help find bugs (see “Testing” lecture)
- Internal test also can help eliminate bugs
  - Validating parameters & checking invariants can eliminate some functions from the bug hunt

**Display Output**

Write values of important variables at critical spots

- Poor:
  ```c
  printf("%d", keyvariable);
  ```
  stdout is buffered; program may crash before output appears

- Maybe better:
  ```c
  printf("%d\n", keyvariable);
  ```
  Printing '\n' flushes the stdout buffer, but not if stdout is redirected to a file

- Better:
  ```c
  printf("%d", keyvariable);
  fflush(stdout);
  ```
  Call fflush() to flush stdout buffer explicitly

**Agenda**

(1) Understand error messages
(2) Think before writing
(3) Look for common bugs
(4) Divide and conquer
(5) Add more internal tests
(6) Display output
(7) Use a debugger
(8) Focus on recent changes
Agenda

(1) Understand error messages
(2) Think before writing
(3) Look for common bugs
(4) Divide and conquer
(5) Add more internal tests
(6) Display output
(7) Use a debugger
(8) Focus on recent changes

Use a Debugger

Use a debugger

• Alternative to displaying output

The GDB Debugger

GNU Debugger
• Part of the GNU development environment
• Integrated with Emacs editor
• Allows user to:
  • Run program
  • Set breakpoints
  • Step through code one line at a time
  • Examine values of variables during run
  • Etc.

For details see precept tutorial, precept reference sheet, Appendix 1

Agenda

(1) Understand error messages
(2) Think before writing
(3) Look for common bugs
(4) Divide and conquer
(5) Add more internal tests
(6) Display output
(7) Use a debugger
(8) Focus on recent changes

Focus on Recent Changes

Focus on recent changes

• Corollary: Debug now, not later

Difficult:
(1) Compose entire program
(2) Test entire program
(3) Debug entire program

Easier:
(1) Compose a little
(2) Test a little
(3) Debug a little
(4) Compose a little
(5) Test a little
(6) Debug a little
...

Focus on Recent Changes (cont.)

• Corollary: Maintain old versions

Difficult:
(1) Change code
(2) Note new bug
(3) Try to remember what changed since last version

Easier:
(1) Backup current version
(2) Change code
(3) Note new bug
(4) Compare code with last version to determine what changed
Maintaining Old Versions

To maintain old versions...

Approach 1: Manually copy project directory

```bash
$ mkdir myproject
$ cd myproject
  Create project files here.

$ cd ..
$ cp -r myproject myprojectDateTime
$ cd myproject
  Continue creating project files here.
```

Approach 2: Use a Revision Control System such as subversion or git

- Allows programmer to:
  - Check-in source code files from working copy to repository
  - Commit revisions from working copy to repository
    - saves all old versions
  - Update source code files from repository to working copy
    - Can retrieve old versions
  - Appropriate for one-developer projects
  - Extremely useful, almost necessary for multidivider projects!

Not required for COS 217, but good to know!

Google "subversion svn" or "git" for more information.

Summary

General debugging strategies and tools:

1. Understand error messages
2. Think before writing
3. Look for common bugs
4. Divide and conquer
5. Add more internal tests
6. Display output
7. Use a debugger
   - Use GDB!!!
8. Focus on recent changes
   - Consider using RCS, etc.

Appendix 1: Using GDB

An example program
File testintmath.c:

```c
#include <stdio.h>
int gcd(int i, int j) {
  int temp;
  while (j != 0) {
    temp = i % j;
    i = j;
    j = temp;
  }
  return i;
}

int lcm(int i, int j) {
  return (i / gcd(i, j)) * j;
}
```

Euclid’s algorithm; Don’t be concerned with details

The program is correct

But let’s pretend it has a runtime error in gcd()...

General GDB strategy:

- Execute the program to the point of interest
  - Use breakpoints and stepping to do that
- Examine the values of variables at that point

Typical steps for using GDB:

(a) Build with –g
gcc217 -g testintmath.c -o testintmath
  - Adds extra information to executable file that GDB uses
(b) Run Emacs, with no arguments
    emacs
(c) Run GDB on executable file from within Emacs
    <Esc key> x gdb <Enter key> testintmath <Enter key>
(d) Set breakpoints, as desired
    break main
    - GDB sets a breakpoint at the first executable line of main()
    break gcd
    - GDB sets a breakpoint at the first executable line of gcd()
Appendix 1: Using GDB

Typical steps for using GDB (cont.):
(e) Run the program
   run
   • GDB stops at the breakpoint in main()
   • Emacs opens window showing source code
   • Emacs highlights line that is to be executed next
   continue
   • GDB stops at the breakpoint in gcd()
   • Emacs highlights line that is to be executed next
(f) Step through the program, as desired
   step (repeatedly)
   • GDB executes the next line (repeatedly)
   • Note: When next line is a call of one of your functions:
   • stop command stops into the function
   • next command steps over the function, that is, executes the next line
   without stepping into the function

Typical steps for using GDB (cont.):
(g) Examine variables, as desired
   print i
   print j
   print temp
   • GDB prints the value of each variable
(h) Examine the function call stack, if desired
   where
   • GDB prints the function call stack
   • Useful for diagnosing crash in large program
(i) Exit gdb
   quit
(j) Exit Emacs
   <Ctrl-x key> <Ctrl-c key>

GDB can do much more:
• Handle command-line arguments
  run arg1 arg2
• Handle redirection of stdin, stdout, stderr
  run < somefile > someotherfile
• Print values of expressions
• Break conditionally
• Etc.